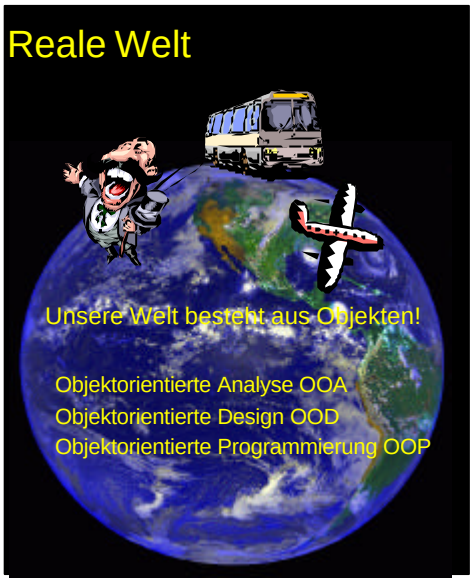
Objektorientierte Programmierung



Virtuelle Welt

- Objekte
  - mit ähnlichen Eigenschaften (Attributen) und
  - ähnlichen Verhaltensweisen (Methoden, „Member Functions“, Elementfunktionen)

  fasst man in Klassen zusammen.
- Ein Objekt aus einer Klasse nennt man eine Instanz der Klasse.

Breymann_Folien

Objektorientierte Programmierung

- Beim Entwurf einer Klasse muss man zunächst die Frage beantworten,
  - welche gemeinsamen Eigenschaften (Attribute) die Objekte der Klasse besitzen und
  - welche gemeinsamen Verhaltensweisen (Methoden) die Objekte der Klassen zeigen bzw.
  - mit welchen Methoden man auf den Objekten der Klasse operieren möchte.
- Ein abstrakter Datentyp (ADT) fasst die Daten (Eigenschaften) und die Funktionen (Methoden), mit denen die Daten bearbeitet werden dürfen zusammen.
- Mit Funktion ist hierbei nicht die konkrete Implementierung, sondern nur die Spezifikation der Zugriffsoperation gemeint.
- Eine Klasse ist ein abstrakter Datentyp, der in einer Programmiersprache formuliert ist.

Breymann_Folien

Abstrakter Datentyp und Datenkapselung

3

- Um unzulässige Zugriffe und damit auch versehentliche Fehler zu vermeiden, sollte die tatsächliche Implementierung der Datenstrukturen einer Klasse nach außen nicht sichtbar sein:

  Schlüsselwort: private

- Man beachte: Die öffentliche Schnittstelle einer jeden Klasse sollte so klein wie möglich gehalten werden, d.h., alle internen Datenstrukturen sollten vor den Anwendern (Nutzern) einer Klasse versteckt werden. Diese Attribute sollten mit dem Schlüsselwort „private“ deklariert werden.

- Nur mittels „öffentlicher“ Elementfunktionen und über deren wohldefinierte Schnittstellen sollte ein Zugriff auf die Daten möglich sein. Diese sollten als „public“ deklariert werden.

  Schlüsselwort: public

Breymann_Folien

Strukturierte Datentypen

4

- Um logisch zusammengehörende Daten von verschiedenen Datentypen zusammenfassen zu können, stellt C/C++ die Struktur(en) zur Verfügung:

```
enum Geschlecht { weiblich, maennlich };
enum Studienfach { Informatik, Bioinformatik, CuK, Computerlinguistik,
                   AngewInformatik};

struct Student {
        string          name;
        Geschlecht      geschlecht;
        unsigned short  semesterzahl;
        Studienfach     studienfach;
        unsigned long   matrikelnummer;
        unsigned short  uebungsnummer;
        string          name_des_bremsers;
        vector<int>     resultate_der_uebungen;
        float           note;
};
```

Breymann_Folien

Strukturen und Klassen

- Strukturen sind Klassen ohne „Privatsphäre“, alle Datenstrukturen sind als „public“ deklariert.
- In der Regel besitzen Strukturen keine Methoden. Die Definition und Deklaration von Methoden ist jedoch möglich.

```
class Student {
        public:
                string          name;
                Geschlecht      geschlecht;
                unsigned short  semesterzahl;
                Studienfach     studienfach;
                unsigned long   matrikelnummer;
                unsigned short  uebungsnummer;
                string          name_des_bremsers;
                vector<int>     resultate_der_uebungen;
                float           note;
        };
```

- Warnung: Das Design der obigen Klasse ist miserabel! Dies ist nur ein Beispiel für die Beziehung zwischen Strukturen und Klassen. Wir werden diese Klasse nach und nach optimieren.

Strukturen und Klassen

- Ein erster Schritt zur Verbesserung besteht darin, die internen Datenstrukturen als „private“ zu deklarieren und eine saubere Schnittstelle für die Benutzer der Klasse durch entsprechende öffentliche Elementfunktionen zu definieren.

```
class Student {
        public:
                //  Hier werden alle für die Anwendung notwendigen Methoden aufgelistet.
                //  Wir geben aus Platzgründen nur drei dieser Methoden an:
                int     ausgabeZahlDerUebungsPunkte() ;   // Gibt die Gesamtzahl zurück
                string ausgabeNamen() ;                   //  Gibt den Namen aus
                void   eingabeNamen(string name_des_studenten);   // Setze den Namen
        private:
                string          name;
                Geschlecht      geschlecht;
                unsigned short  semesterzahl;
                Studienfach     studienfach;
                unsigned long   matrikelnummer;
                unsigned short  uebungsnummer;
                vector<int>     resultate_der_uebungen;
                float           note;
};
```

Strukturen und Klassen

- Der Entwickler einer Klasse implementiert die Elementfunktionen und passt diese an, wenn er die internen Datenstrukturen ändert oder wenn er einen besseren Algorithmus einsetzten möchte.
- Die Schnittstellen (Namen der Elementfunktionen und die Parameter) sollten sich jedoch nicht ändern.
- Änderungen der obigen Art erfordern vom Nutzer nur, dass er das Programm neu übersetzt. Er muss jedoch seine Programme nicht anpassen.
- Neue sinnvolle Elementfunktionen können natürlich im Verlauf der Weiterentwicklung einer Klasse hinzukommen.

Klassen

- Eine C++ Klasse hat folgende typische Gestalt:

```
class Klassenname {
   public:
     Rückgabetyp Elementfunktion1(Parameterliste1);
     Rückgabetyp Elementfunktion2(Parameterliste2);
     // ... weitere Elementfunktionen
   private:
     Datentyp Attribut1;
     Datentyp Attribut2;
     //  .... weitere Attribute
};
```

- Wir werden die Details der Syntax einer Klassendefinition nach und nach kennen lernen.

Eine erste Klasse

9

- Wir studieren nun eine erste Klasse, die im Breymann-Skript unter dem Namen ort1_h eingeführt wird und die Punkte in der Ebene repräsentiert.
- Wir werden im folgenden englische Namen verwenden, da im englischen die Namen der Attribute und Methoden in der Regel kürzer sind.

```
// header of class Point2D: point2D.h
#ifndef point2D_h
#define point2D_h

class Point2D {
        public:
                int   X() const;   // Das Schlüsselwort „const“ drückt aus, dass die damit
                int   Y() const;   //  markierten Elementfunktionen Objekte nicht verändert.
                void   changeCoordinates( int x, int y);   // x,y = neue Werte
        private:
                int  xCoordinate;
                int  yCoordinate;
    };
#endif     // point2D_h
```

Breymann_Folien

Eine erste Klasse

10

- Die Header-Datei einer Klasse (Beispiel: point2D.h) enthält in der Regel nur die Deklarationen der Elementfunktionen.
- Die Implementierungen der Methoden speichert man in einer Datei, die mit dem Klassennamen (kleiner Anfangsbuchstaben) beginnt und die mit „.cpp“ oder endet:

```
//  Implementierung der Klassenfunktionen von point2D
//  in der Datei „point2D.cpp“
#include “point2D.h“
#include<iostream>
using namespace std;

int Point2D::X( ) const { return xCoordinate; }
int Point2D::Y( ) const { return yCoordinate; }

void Point2D::changeCoordinates(int x, int y) {
    xCoordinate = x;
    yCoordinate = y;
}
```

- Der dem Bereichsoperator „::“ vorgestellte Klassennamen gibt an, dass es sich bei diesen Funktionen um Klassenfunktionen der Klasse „Point2D“ handelt.

Breymann_Folien

Eine erste Klasse

- Nur Elementfunktionen einer Klasse ist der Zugriff auf die privaten Datenstrukturen erlaubt!!!

```
#include “point2D.h“
#include<iostream>
using namespace std;

int Point2D::X( ) const  { return xCoordinate; }
int Point2D::Y( ) const  { return yCoordinate; }

void Point2D::changeCoordinates(int x, int y) {
   xCoordinate = x;
   yCoordinate = y;
}
```

Breymann_Folien

inline-Funktionen

- Funktionsaufrufe kosten Zeit:
  - Der Zustand des aufrufenden Programms muss gesichert werden und
  - Parameter müssen übergeben und eventuell kopiert werden.
- Dieser Aufwand kann reduziert werden, in dem man Funktionen als „inline“-Funktionen deklariert:

```
inline int Point2D::X( );
inline int Point2D::Y( );
inline void Point2D::changeCoordinates(int x, int y);
```

- „inline“ bewirkt, dass beim Compilieren der Aufruf durch den Funktionskörper ersetzt wird. Beispiel:

```
inline int quadrat(int x) {  return x*x; }
```

Ein Aufruf der Form: `z = quadrat(100);` wird vom Compiler durch `z = 100 * 100;` ersetzt.

- Die Verwendung von „inline“ kann nur für kurze Funktionen empfohlen werden.

Breymann_Folien

inline-Funktionen

13

- Die Programmierung mit Klassen verwendet viele kleine Funktionen, so dass sich die Verwendung von „inline“-Funktionen lohnt.
- Es gibt drei Möglichkeiten inline-Funktionen zu definieren, von denen wir nur zwei empfehlen können:
- 1. Innerhalb der Klasse:

```
class Point2D {
        public:
                int   X() const { return xCoordinate; }
                int   Y() const { return yCoordinate; }
                void   changeCoordinates ( int x, int y) {
                        xCoordinate = x;
                        yCoordinate = y;
                }
        private:
                int xCoordinate;
                int yCoordinate;
    };
```

Breymann_Folien

inline-Funktionen

14

- 1. Innerhalb der Header-Datei:

```
class Point2D {
        public:
                int   X() const;
                int   Y() const;
                void   changeCoordinates ( int x, int y);
        private:
                int xCoordinate;
                int yCoordinate;
    };
  // ------- zweite inline Implementierung ---------
    inline int   X() const { return xCoordinate; }
    inline int   Y() const { return yCoordinate; }
    inline void   changeCoordinates ( int x, int y) {
                        xCoordinate = x;
                        yCoordinate = y;
    }
```

Breymann_Folien

Anwendung der Klasse Point2D

- Einfaches Beispielprogramm:

```
// Anwendung der Klasse Point2D in Datei main.cpp
#include "point2D.h"
#include<iostream>
using namespace std;

int main( ) {
   Point2D  firstPoint;                          //  Ein Objekt erzeugen
   firstPoint.changeCoordinates(100, 200);   // Koordinaten ändern
   cout  <<  "Der Punkt hat die Koordinaten"
         <<  " x = "  <<  firstPoint.X()
         <<  " y = "  <<  firstPoint.Y()  << endl;
}
```


Konstruktoren

- Da Klassen nur Objekte beschreiben, müssen zunächst Objekte der Klasse (Instanzen) erzeugt werden:

```
Point2D  firstPoint;      // erzeuge Objekt „firstPoint"
```

- Die Objekterzeugung entspricht der bekannten Variablen-definition, wobei die Klasse den Datentyp darstellt.
- Beim Ablauf des Programms wird an der Stelle der Definition des Objekts eine besondere Klassenfunktion, der so genannte Konstruktor aufgerufen.
- Der implizite Aufruf des Konstruktor generiert das Objekt „firstPoint" und stellt Speicherplatz zur Verfügung.
- Der Konstruktor wurde in unserem ersten Beispiel vom System zur Verfügung (Default-Konstruktor) gestellt.
- Er kann aber auch selbst definiert (implementiert) werden.

Konstruktoren

17

- Allgemeine Konstruktoren können im Gegensatz zu Default-Konstruktoren Argumente haben.

```
Point2D::Point2D(int x, int y ) {    xCoordinate = x;  yCoordinate = y; }
```

- Konstruktoren können wie andere Funktionen überladen werden, d.h., dass es mehrere allgemeine Konstruktoren mit unterschiedlichen Parameterlisten geben kann:

```
Point2D::Point2D( int x  ) {    xCoordinate = x;  yCoordinate = 200; }
```

- Der Compiler wählt dann den passenden Konstruktor aus, in dem er Anzahl und Datentypen der Argumente der Parameterliste der Konstruktorendeklaration mit der Angabe im Aufruf vergleicht:

```
int x = 10, y = 20;
Point2D  firstPoint;              //  Default-Konstruktor
Point2D  secondPoint(x);          //  siehe Mitte
Point2D  thirdPoint(50, y);       //  siehe oben
```

- Man beachte: Wird ein allgemeiner Konstruktor deklariert und definiert, so muss auch der Default-Konstruktor deklariert und definiert werden.

Breymann_Folien

Konstruktoren

18

- Was geschieht beim Aufruf eines Konstruktors:
  - Zunächst wird Speicherplatz für die Datenelemente reserviert:
    - xCoordinate und
    - yCoordinate
  - Dann werden die Aktualparameter zugewiesen und
  - schließlich wird der Programmcode innerhalb der geschweiften Klammern ausgeführt.
- Eine effiziente Art der Initialisierung bietet die so genannte Initialisierungs-liste, die vor dem eigentlichen Funktionscode ausgeführt wird:

```
Point2D::Point2D(int x, int y )
: xCoordinate(x), yCoordinate(y)      // Initialisierungsliste
{
   // in diesem Beispiel : leerer Block
}
```

- Man beachte: Die Reihenfolge der Initialisierung richtet sich nach der Reihenfolge in der Deklaration der Klasse !!!

Breymann_Folien

Konstruktoren

- Ein besonderer Konstruktor ist der Kopierkonstruktor (copy constructor).
- Er dient der Initialisierung eines Objektes mit den Daten eines anderen Objekts.
- Der Kopierkonstruktur hat als Argument eine Referenz auf ein Objekt der entsprechenden Klasse:

```
Point2D::Point2D(const Point2D & pointToCopy)
: xCoordinate(pointToCopy.xCoordinate),
  yCoordinate(pointToCopy.yCoordinate)
{  // auch  in diesem Beispiel : leerer Block
}
```

- Der Kopierkonstruktor wird nur dann verwendet, wenn ein neues Objekt generiert wird:

```
Point2D  firstPoint(19,39);
Point2D  secondPoint = firstPoint;
```

- Man beachte: Im obigen Beispiel wird der Kopierkonstruktor verwendet (nicht der Zuweisungsoperator, siehe auch nächste Seite).

Breymann_Folien

Konstruktoren


```
Point2D   p1, p2;              //  Default-Konstruktor
p1  =  Point2D(8,7);           //  allgemeiner Konstruktor + Zuweisungsoperator
p2  =  p1;                     //   Zuweisung
Point p3  = Point2D(1, 17);    // ein temporäres Objekt wird in das neu erzeugte
                               //  Objekt kopiert.
```

- Die Übergabe von Objekten an eine Funktion per Wert und die Rückgabe eines Ergebnisobjekts wird ebenfalls als Initialisierung betrachtet, d.h., hier wird der Kopierkonstruktur implizit aufgerufen.

Breymann_Folien

Konstruktoren

- Die Klassenbeschreibung von Point2D mit allen angegebenen Konstruktoren hätte die folgende Form:

```
class Point2D {
        public:
                Point2D( );                             // Default-Konstruktor
                Point2D(int x);                         // Allg. Konstruktor 1
                Point2D(int x, int y);                  // Allg. Konstruktor 2
                Point2D(Point2D  &pointToCopy);         // Kopierkonstruktoren
                int   X() const;
                int   Y() const;
                void   changeCoordinates(int x, int y);
        private:
                int  xCoordinate;
                int  yCoordinate;
    };
```


Zuweisungsoperator =

- Falls kein spezieller Zuweisungsoperator in einer Klasse definiert ist, wird automatisch vom Compiler ein Zuweisungsoperator bereitgestellt.
- Hierbei wird für jedes Element, das selbst Objekt oder aber von einem Grunddatentyp ist, der zugehörige Zuweisungsoperator aufgerufen.
- Beispiel für einen Zuweisungsoperator der Klasse Point2D:

```
Point2D& Point2D::operator=(const Point2D & secondPoint) {
    if (this != &secondPoint) {   // Zuweisung identischer Objekte vermeiden
      this->xCoordinate = secondPoint.xCoordinate;
      this->yCoordinate = secondPoint.yCoordinate;
    }
    return *this;
}
```

- this-Zeiger: this ist ein Schlüsselwort, das innerhalb einer Elementfunktion einen Zeiger auf das Objekt darstellt:
  - this ist ein Zeiger auf das aktuelle Objekt.
  - *this ist das Objekt selbst.

Destruktoren

- Destruktoren sind Elementfunktionen, die den Speicherplatz von nicht mehr benötigten Elementen wieder freigeben.
- Falls kein Destruktor für eine Klasse deklariert und definiert wird, erzeugt der Compiler automatisch einen Destruktor.
- Automatisch erzeugte Destruktoren von Klassen mit dynamischen Strukturen geben in der Regel nicht den gesamten Speicherplatz wieder frei (memory leaks).
- Destruktoren besitzen keine Argumente und keinen Rückgabetyp. In der Deklaration wird ein Tilde ~ vorangestellt:

```
class Point2D {
        public:
                Point2D( );                          // Default-Konstruktor
                Point2D(int x);                      // Allg. Konstruktor 1
                Point2D(int x, int y);               // Allg. Konstruktor 2
                Point2D(Point2D &pointToCopy );      // Kopierkonstruktoren
                ~Point2D( );                         // Destruktor
                int  X() const;
                int  Y() const;
                void changeCoordinates(int x, int y);
        private:
                int xCoordinate;
                int yCoordinate;
    };
```

Breymann_Folien

Destruktoren und Gültigkeitsbereiche

- Die Gültigkeit oder Lebensdauer eines Objekts bzw. einer Variablen endet, an dem mit der schließenden geschweiften Klammer markierten Ende des Blocks, in dem das Objekt oder die Variable definiert wurde.
- Dann wird das Objekt automatisch zerstört, d.h., der Compiler ruft den Destruktor für das Objekt auf.

```
{
   // äußerer Block ....
   Point2D p1;
   // äußerer Block ....
    {
      //  innerer Block mit Anweisungen und Deklarationen
      Point2D p2;
     //  innerer Block          <-- Compiler ruft den Destruktor für p2 auf
    }
   // äußerer Block .....      <-- Compiler ruft den Destruktor für p1 auf
}
```

Breymann_Folien

Destruktoren und Gültigkeitsbereiche

25

- Möchte man den Speicherplatz eines Objekts schon vor dem Ende des entsprechenden Blocks freigeben, so kann man den Destruktor für das Objekt explizit aufrufen.

```
delete p1;
delete p2;
```

- Bei Feldern a[ ] und Vektoren v steht nach dem Schlüsselwort „delete“ noch der Klammeroperator:

```
delete [ ] a;
delete [ ] v;
```

- Variablen, die außerhalb von main und allen anderen Funktionen definiert sind, heißen global.
- Globale Variablen sind in allen Teilen eines Programms gültig.
- Eine globale Variable muss in einer anderen Datei als extern deklariert werden, um dort benutzbar zu sein.
- Man beachte: Die Verwendung von globalen Variablen sollte möglichst vermieden werden (schlecher Programmierstil)!!!

Breymann_Folien

Destruktoren und Gültigkeitsbereiche

26

- Um den Gültigkeitsbereich von Funktionen und Variablen auf bestimmte Dateien zu beschränken, ist das Schlüsselwort „static“ notwendig.

```
// Datei1.cpp
int global;
int main( ) {
  global = 17;
}

// Datei2.cpp
extern int global;
int func1( ) {
  global = 123;
}
```

```
// Datei1.cpp
static int global;    // nicht mehr „global“
int main( ) {
  global = 17;
}

// Datei2.cpp
static int global;
int func1( ) {
  global = 123;
}
```

Breymann_Folien

Funktionstemplates

- Oft ist die gleiche Aufgabe für verschiedene Datentypen bzw. Klassen zu erledigen.
- Zum Beispiel das Sortieren eines int-Vektors, eines float-Vectors oder eines double-Vectors.
- So genannte Schablonen (Templates) erlauben C++ Nutzern Funktionen mit parametrisierten Datentypen zu schreiben.
- Für den noch unbekannten Datentyp wird ein Platzhalter (Parameter) eingefügt.
- Die allgemeine Form für Templates ist:

template<class Typbezeichner> Funktionsdefinition

Breymann_Folien

Funktionstemplates

- Beispiel: Sortieren mit Mergesort.

```
//mergesort.cpp
#include<iostream>
#include<vector>
#include<cstddef>
using namespace std;

template<class T>
void merge(vector<T>& a, int left, int middle, int right) {
    vector<T> help;    // Vektor zum Zwischenspeichern des sortierten Teilfeldes
      for (int i = left, int j = middle + 1; i <= middle && j <= right; ) {
          for ( ; a[i] <= a[j]; ++i)  help.push_back(a[i]);
          for ( ; a[i]  > a[j]; ++j)  help.push_back(a[j]);
      }
      for ( ; i <= middle; ++i)  help.push_back(a[i]);        // Rest des linken Felds
      for ( ; j <= right; ++j)  help.push_back(a[j]);         // Rest des rechten Felds

      for (int i = 0, int j = left; j <= right; ++i, ++j)  a[j] = help[i];
      delete [ ] help;
}
```

Breymann_Folien

Funktionstemplates

- Beispiel: Sortieren mit Mergesort.

```
template<class T>
void mergesort(vector<T>& a, int left, int right) {
   int middle = (right – left)/2;
   if (middle >=1) {
     middle += left;
     mergesort(a, left, middle);
     mergesort(a, middle+1, right);
     merge(a, left, middle, right);
   }
}
```

Breymann_Folien